GPGPU computation and visualization of three-dimensional cellular automata

Gobron, S ; Cöltekin, Arzu ; Bonafos, H ; Thalmann, D

Abstract: This paper presents a general-purpose simulation approach integrating a set of technological developments and algorithmic methods in cellular automata (CA) domain. The approach provides a general-purpose computing on graphics processor units (GPGPU) implementation for computing and multiple rendering of any direct-neighbor three-dimensional (3D) CA. The major contributions of this paper are: the CA processing and the visualization of large 3D matrices computed in real time; the proposal of an original method to encode and transmit large CA functions to the graphics processor units in real time; and clarification of the notion of top-down and bottom-up approaches to CA that non-CA experts often confuse. Additionally a practical technique to simplify the finding of CA functions is implemented using a 3D symmetric configuration on an interactive user interface with simultaneous inside and surface visualizations. The interactive user interface allows for testing the system with different project ideas and serves as a test bed for performance evaluation. To illustrate the flexibility of the proposed method, visual outputs from diverse areas are demonstrated. Computational performance data are also provided to demonstrate the method's efficiency. Results indicate that when large matrices are processed, computations using GPU are two to three hundred times faster than the identical algorithms using CPU.

DOI: https://doi.org/10.1007/s00371-010-0515-1

The Visual Computer manuscript No.
(will be inserted by the editor)

GPGPU Computation and Visualization of Three-dimensional Cellular Automata

Stéphane Gobron · Arzu Çöltekin · Hervé Bonafos · Daniel Thalmann

Abstract This paper presents a general-purpose simulation approach integrating a set of technological developments and algorithmic methods in cellular automata (CA) domain. The approach provides a *general-purpose computing on graphics processor units* (GPGPU) implementation for computing and multi-rendering of any direct-neighbor three dimensional (3D) CA. The major contributions of this paper are; the CA processing and the visualization of large 3D matrices computed in real-time; the proposal of an original method to encode and transmit large CA functions to the graphics processor units in real time; and clarification of the notion of *top-down* and *bottom-up* approaches to CA that non-CA experts often confuse. Additionally a practical technique to simplify the finding of CA functions is implemented using a 3D symmetric configuration on an interactive user interface with simultaneous inside and surface visualizations. The interactive user interface allows for testing the system with different project ideas and serves as a test bed for performance evaluation. To illustrate the flexibility of the proposed method, visual outputs from diverse areas are demonstrated. Computational performance data are also provided to demonstrate the method's efficiency. Results indicate that when large matrices are processed, computations using GPU are two to three hundred times faster than the identical algorithms using CPU.

Keywords Cellular automata, GPGPU, Simulation of natural phenomena, Emerging behavior, Volume graphics, Information visualization, Real-time rendering, Medical visualization

S. Gobron (cor.)
EPFL, IC, ISIM, VRLAB, Station 14, CH-1015 Lausanne, Switzerland. E-mail: stephane.gobron@epfl.ch

A. Çöltekin
University of Zürich, GIVA, Department of Geography, Winterthurerstr. 190, CH-8057 Zurich, Switzerland.

H. Bonafos
Tecnomade, 41 place Carrière, 54000 Nancy, France.

D. Thalmann
EPFL, IC, ISIM, VRLAB, Station 14, CH-1015 Lausanne, Switzerland.

1 Introduction

Cellular automata (CA) allow efficient computations in a wide variety of fields including simulation of natural phenomena and physical processes (*e.g.* [8,13,12], [28]). CA algorithms can be even faster and more powerful when run on a graphics processing unit (GPU) [29], particularly when the input is large [11,21,25]. An approach that utilizes GPU accelerated real-time visualizations to identify emerging phenomena for two dimensional (2D) CA has previously been suggested [10]. The 2D approach uses hexagonal grids and is further discussed in section 2.2. The study proposed in this paper extends the 2D approach by exploring the GPU based computation and rendering of real-time Boolean multi-dimensional CA and implements in three-dimensional (3D) voxel space. An overview of the user interface and several example outputs can be seen in Figure 1. This graphical user interface serves as a test bed for performance testing, benchmarking and CA related project development.

Approaches that offer stability and speed at low computational costs have long been desirable for real-time 3D visualizations [2,4,11] as well as real-time large inter-cellular computations. An important aspect of this study is to help progress in this area by transferring

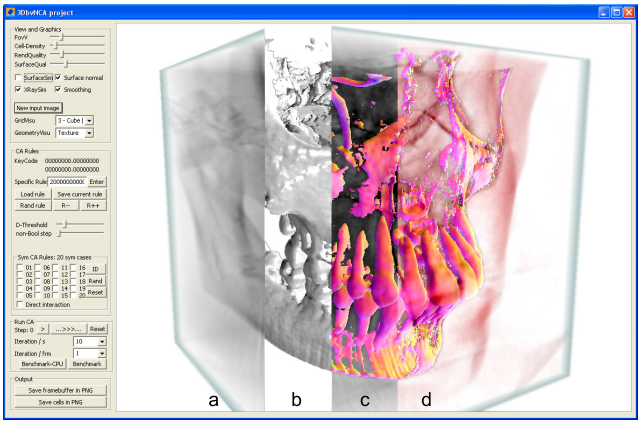

Fig. 1 A composite of four screen shots of the test bed that were created to evaluate the method: (a) X-ray-like rendering, (b) surface rendering, (c) simultaneous X-ray and vectorial surface rendering, and (d) simultaneous rendering and 3D CA computational interaction on the voxel structure. The implementation is provided as a freeware and can be obtained at *http://vrlab.epfl.ch/~stephane/*

massive computations and multiple rendering of CA processes to the GPU (see Section 4). Several interdisciplinary communities will benefit from the proposed method for identifying formal CA families more efficiently. High performance gain we provide may also encourage the computer graphics community to take advantage of our approach to identify and use specific CA rules for simulation of complex phenomena or geometric transformations. Using this approach, other scientific communities dealing with 3D matrices (such as magnetic resonance imaging (MRI) in bio-medical applications) can process and visualize their large datasets in real-time on a common personal computer.

The contributions of this paper include:

A. A test bed for CA experts: - The processing of a 3D regular grid CA using up to 17 million cells *almost instantaneously* (*i.e.* less than $10^{-}3$ second), - The 3D visualization of large 3D matrices in real-time (*i.e.* superior to 60 fps), - Access to a free software demonstrating the test bed using modern and common personal computers (for Microsoft Windows).

B. Progress concerning coding and understanding CA in the world of computer graphics: - The proposal of an original method to encode and transmit large CA functions to the GPUs *in real time*; - Clarification of the notion of *top-down* and *bottom-up* approaches that non-CA experts often confuse; - A practical technique to simplify the finding of CA functions, using 3D symmetric configuration on an interactive user interface with simultaneous interior and surface visualizations.

The remainder of the paper is organized as follows. Section 2 offers a brief overview of the state of the art in the field. Section 3 documents concepts on GPU as well as terminologies on CA. Section 4 defines CA concepts and presents CA processes, and Section 5 explains the approach allowing the computation of 3D CA using the graphics card. This is followed by Sections 6 and 7 presenting the results and conclusions.

2 Background

2.1 Definition and a brief history of CA

The concept of CA was invented in the early 1950's by J. Von Neumann reportedly upon Stan Ulam's suggestion [14]. Cellular automata can be described in several ways, but in the most generic sense, it can be said that CA *model and mimic life* as simulations. While mimicking life was perhaps the original motivation, CA should be considered more as simulators for dynamic systems –which is more general than life-like behavior.

Appropriately, probably the best-known CA simulation is Conway's "Game of Life" [3] a very simple rule set that exhibits a wide range of complex behavior –first applied to CG in [24]. This simplistic and yet complex model is often viewed as the source of public awareness of cellular automata, however it was Gardner who popularized it with his early 1970s publications [7,8]. The next milestone in CA history is Wolfram's classification of automata [26,27]. Among the more recent research, the study of excitable media is one domain that attracted wide attention, where CA models have been found useful for approximating real life behavior. Gerhardt *et al.* [9] designed a CA model that adheres to the curvature and dispersion properties found experimentally in excitable media [25]. A detailed survey on CA up to this date is presented by Ganguly *et al.* [6]. Since then, the *computational era* has provided researchers with tools and infrastructure, of which a wide variety of interesting CA implementations can be found in literature, however the publications are too numerous to mention within the scope of this paper.

2.2 Bottom-up and top-down approaches

When studying the CA literature, the reader may feel confused about what appears to be two distinct approaches towards CA:

– *bottom-up* approach studies the local behavior for the purpose of finding direct logic based rules. To reach a simplified model of the original problem, a discrete system is selected that inherently possesses some necessary basic properties *e.g.* symmetric structures (see section 4.1);

Table 1 Summary of the advantages and disadvantages of *bottom-up* or *top-down* approaches.

	advantage	disadvantage
Top-down	- Research topic is clearly defined - Formulated mathematical equations	- Discretization method produces error - CA key-code is almost impossible to find - Combining top-down derived CA rules is very difficult and often, a new top-down approach must be developed
Bottom-up	- CA key-code derivation methods are stable and can be repeated - Algorithm is faster / the most efficient - Key-codes can be combined (multi-phenomena)	- Cannot find CA key-code from a specific topic - Cannot easily find mathematical equation

– *top-down* approach derives (most of the time based on poor and non-optimized discretization of mathematical function) local rules based on the specific behavior of a phenomena. It can be defined as the transformation of a differential equation into a discrete system that can be simulated on a computer.

In the literature and especially in computer graphics texts, most examples seem to use the top-down approach and have been accepted as such with the term CA associated with it. Top-down methods are efficient for finding an approximation of a precise phenomenon (*e.g.* melting). Often resulting in elegant but typically inflexible and often unstable solutions, as they are almost impossible to combine with others; for instance, wood on fire, heat waves, ash production, and melting of the snow around the campfire.

In this paper, we propose a bottom-up approach to find the CA rules and determine useful global behavior. Predicting emerging phenomena using CA has been a challenging task and often impossible based on theoretical approaches [27]. Hence, bottom-up approaches do not guarantee *when* the CA rule corresponding to such phenomena will be found.

In our approach, the search process is supported with a set of real-time visualizations using transparencies and implicit surfaces to help user interpreting the CA behavior. Furthermore, for symmetrical CA, a set of geometric tools facilitates a logical crystalline construction that enables the test bed to behave as a tool box. Based on these properties, we experimentally create CA functions until the visualization gives the desired result. When found, such formal CA functions are simple to apply, extremely efficient in terms of computational cost, and combinations between different types of behavior are possible. These properties offer potential for multiple simulations of complex phenomena to interact with each other (*e.g.,* complex multiple behaviors of the burning of solid objects is possible all in one model).

Depending on the task and/or application, it is difficult to suggest which of the *bottom-up* or *top-down* approaches is the more efficient approach. We consider both approaches to have their own unique advantages and disadvantages (Table 1). Nevertheless, an awareness of the difference between them is necessary for understanding CA at a fundamental level, and therefore for understanding the contribution of this paper. *Top-down* will converge to an approximation of the solution that has almost no chance of being the CA key-code (for a definition of a key-code, please refer to Section 3.2), and that will be almost impossible to combine with other behaviors. The *bottom-up* solution is tedious at first, but results into *the* unique solution. When more than one CA key-code are found, they can easily be combined. Table 1 summarizes advantages and disadvantages of both approaches.

2.3 CA and 3D on GPU

Compared to the large number of 3D graphics applications on GPU (*e.g.,* [4,5]), top-down or buttom-up CA implementations on GPU are still rare ([21], [30],[10], [29]), and none were found in the literature for generic real-time bottom-up 3DCA. Current GPU implementations (*e.g.,* [23], [30], [29]) use top-down approaches and differ from our approach from this point of view. A paper by Harris *et al.* [11] is perhaps one of the most interesting early publications reporting a top-down implementation on GPU that is relevant to this study. They implemented an extension of CA called a *coupled map lattice* (CML) on early programmable graphics hardware. They achieved about 25 times faster processing over a roughly equivalent CPU implementation. While Harris *et al.* [11] paper is valuable in terms of an early demonstration of the capabilities of graphics hardware, the approach in our work is significantly different. Our study provides a generic and bottom-up implementation.

3 GPU and CA concepts

Based on an introduction in [10], this section summarizes concepts relevant to GPU and CA.

3.1 Graphics processor unit (GPU)

GPU performance has increased dramatically over the last four years when compared to CPUs [15], and more recently an improvement of up to 900 times was demonstrated [10]. The performance related reports show varying numbers for how much acceleration is gained. This is due to the fact that results largely depend on the graphics card models and the operating system. Nevertheless, in all cases there is a significant and consistent improvement in the reported performances. To achieve this, the graphics cards typically use three programs in their pipeline: the *Vertex*, the *Geometric*, and the *Fragment Shaders* also called *Pixel Shaders*. These three programs are illustrated in the right of Figure 3(b). To use the graphics card as a computational device for CA, only the last program is important as it performs computations on single 2D textures that are used to store the 3D cellular matrix. A detailed survey on GPU architecture is not in the scope of this current work, especially since we focus on CA-function deduction and their CG applications. For an exhaustive review on *general-purpose processing on graphics processor unit* (GPGPU) the reader is recommended to refer to the Owens *et al.* [15] survey.

3.2 CA terminology, concepts and definitions

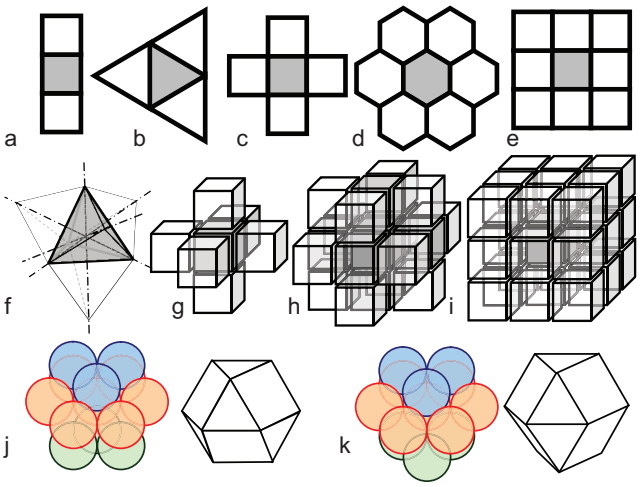

Fig. 2 Multidimensional geometric representations appropriate for CA: (a) $1D$; (b) $2D^t$; (c) $2D^{vN}$; (d) $2D^h$; (e) $2D^M$; (f) $3D^t$; (g) $3D^{vN}$; (h) $3D^{M_1}$; (i) $3D^{M_2}$; (j) $3D^{h_1}$; (k) $3D^{h_2}$

This section presents the terminology used in this paper including various notions of CA types, neighborhoods, rules, changes of state, and CA-key codes. Figure 2 illustrates samples of structures that present geometric properties particularly suitable for CA use. The first row proposes symmetric geometric representations for the first and second dimensions. The second row introduces the four most simple symmetric CA geometric structures for the third dimension respectively based on tetrahedron (f), and voxel-like structures (g,h,i). The bottom row presents the two possible representations of what is probably the simplest of the natural 3D structures, *i.e.* "close packing", with its cellular 3D shape. These are the hexagonal close packing (j) such as platinum atoms and in the cubic close packing (k) such as silver, gold, copper, and lead atoms [16]. Considering the multitude of possible CA structures and types, the following rule is used to symbolize them all:

$$[\text{dimension}]D^{[\text{structure}]}_{[\text{state nb}]}CA \equiv dD^N_nCA \tag{1}$$

With:

- d as dimension;
- n the number of states that a cell C can have;
- N the structure:
  - for the square/cube structure with direct neighbors $N \equiv$ 'vN' $= 4$ in 2D, vN stands for **v**on **N**eumann;
  - for the square/cube structure with indirect neighbors $N \equiv$ 'M_i' $= 8$ in 2D, M_i stands for **M**oore with i being the growing number of possible indirect cases;
  - for the hexagonal grid $N \equiv$ 'h' $= 6$ in 2D;
  - for the triangle or tetragonal structure $N \equiv$ 't' $= 3$ in 2D and $= 4$ in 3D;

Therefore, in the dD^N_nCA domain, for each cell C with N number of neighbors, let us define c as the number of configuration structures, and ρ as the number of possible CA that can be deduced with:

$$\text{with } c = n^{(N+1)} \text{ and } \rho = n^c \text{ , then } \rho = n^{n^{(N+1)}} \tag{2}$$

For each CA of a specific type, we define a unique *key-code* as φ_1, φ_2,... , φ_c, describing the intrinsic local behavior for each possible state. In fact, a key-code describes the resulting column of the corresponding CA truth table. When using a top-down approach, it is almost impossible to find such a truth table (and therefore its key-code). This is because finding all local logical change of states for each discretized behavior from a general mathematical equation is not feasible.

Let τ be the truth table made with the neighbors of a central cell: $A, B, C...X$. For illustration, this "pivot" cell is located in the center Figure 2g, and Figure 4 depicts all possible symmetrical rotations critically useful for symmetrical CA development and function deduction.

Table 2 Corresponding symmetric shape for the 128 bits $3D_b^{vN}CA$ key-code –*i.e.* the neighbor's state code *Nstate*.

Nstate	Corresponding $3D_b^{vN}CA$ symmetric shape –see Figure 4
0..31	1 2 3 4 3 4 5 7 3 4 5 7 6 8 9 11 3 4 6 8 5 7 9 11 5 7 9 11 9 11 13 15
32..63	3 4 5 7 5 7 10 12 5 7 10 12 9 11 14 16 5 7 9 11 10 12 14 16 10 12 14 16 14 16 17 18
64..95	*identical to previous row*
96..127	6 8 9 11 9 11 14 16 9 11 14 16 13 15 17 18 9 11 13 15 14 16 17 18 14 16 17 18 17 18 19 20

Following section will present the proposed model for computing any rule of $3D_b^{vN}CA$ while enabling the visualizations of the corresponding data flow in real-time.

4 Processes

As can be seen in Figure 3(a) the process cycle consists of three main steps: *input* (first and second steps shown in green), *processes* and *output*. *Input* takes a 3D dataset, a density matrix such as an fMRI scan, which is preprocessed into Tex2D. Tex2D is a 2D representation of *slices*, as most density data are recorded in slices. This allows the viewer to see the volume mapped into a series of slices on a single image.

Several *processes* are integrated in the implementation and the interface (Figure 1) allows for a number of modifications. The experimenter (test bed user) has a great deal of control over the processes. Once the 3D dataset and the desired parameters are provided, the software communicates with the graphics card for real-time CA computations, as specified by the experimenter, and returns real-time visualizations of CA accordingly. During the processes symmetric and asymmetric rules are applied using a floating point variation with visual cellular behavior. All these processes are performed efficiently and they return 3D simulations, Tex2D output, and potentially valuable CA key-codes. For more details on performance, see section 6.2.

4.1 Symmetry rules

By using equation 2, $\rho_{(3D_b^{vN}CA)}$ is equal to 2^{128}, approximately $3.4 \cdot 10^{38}$. To avoid digging randomly into this very large number of possible Boolean CA, this method focuses on symmetric CA. As illustrated by Figure 4, there are 20 possible symmetric configurations. Each of them can be true or false for each CA, therefore, the total number of CA is 2^{20}. This greatly narrows the search space. Focusing on symmetric CA, using crystalline symmetries depicted in Figure 4 and obtaining real-time results (where both surface and interior properties can easily be visualized and interpreted) tremendously increase the chance of converging to interesting CA rules (key-codes).

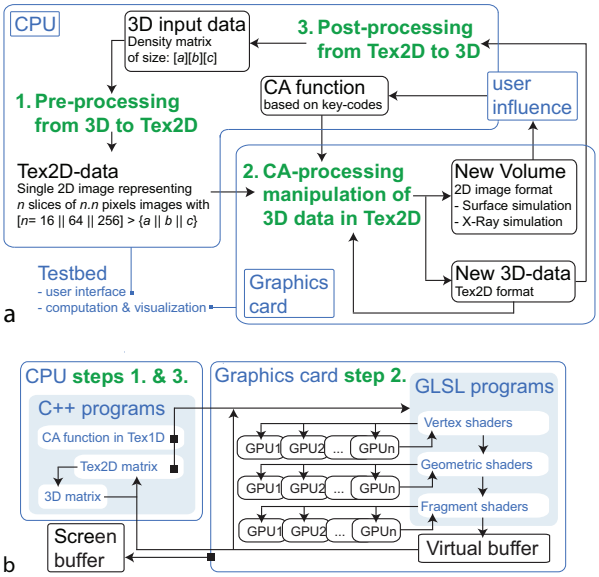

Fig. 3 Data-processing flow diagrams: (a) process cycle for the implementation that allows real-time computation and visualization of any $3D_b^{vN}CA$. (b) relationship between CPU and GPUs –using a simplified representation of the GPUs architecture

As illustrated in [10], symmetric CA can be very useful in the domain of real-time imaging, especially in computer graphics for automatic surface texturing or analysis of image flow.

To determine the relationship between a symmetric CA function and its equivalent key-code (notice that the reciprocity is not valid), we use Table 2 that maps for each of the 128 states and the corresponding 20 symmetrical configurations.

The first practical example based on emerging behaviors is displayed in Figure 5. The behavior depicts a convergence to the bounding box of each object in the scene. To achieve this, we applied a specific symmetric key-code, *i.e.* 1 to 20 possible ϕ' set to *true*, equivalent to a CA-rule with a key-code of 32ϕ.

A second example is finding the derivative (*i.e.* surface normal) of a volume as described by the following key code equation:

$$\varphi'_{true} = [2.4.7.8.11.12.15.16.18.20] \equiv$$
$$\tau(\varphi_{(1..32)} = [AAAAAAAAAAAAAAA2AAAAAAA2AAA2A228])$$

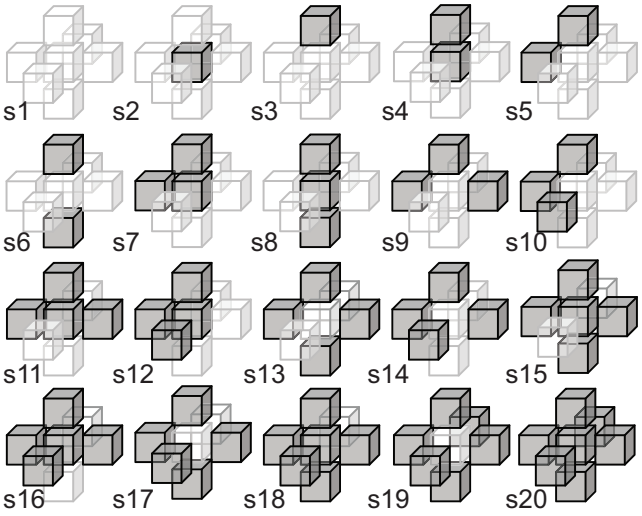

Fig. 4 All symmetric configurations for Boolean direct neighbors CA ($3D_b^{vN}CA$)

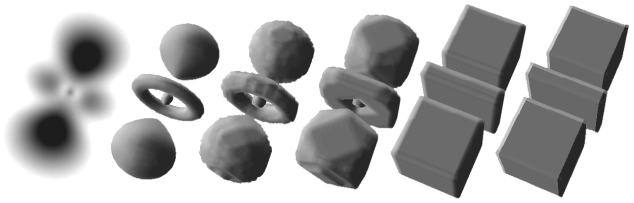

Fig. 5 A simple example of application in geometry using CA: finding bounding boxes on a hydrogen atom simulation database

4.2 Fragment Shaders model

As previously stated, the algorithm of the 3D direct square neighbor CA is based on the relationship between a CPU program and shader codes. To make such a software, three main functions are required: CA computations, the 3DCA rendering, and an interface (also referred to as *human machine interface* or *user interface* for the interactive applications). As the first two functions are computationally highly expensive, both are implemented using shaders. Furthermore, when there are more than six neighbors in a CA implementation, the number of key-codes produced by the software becomes very high and impossible to transmit to GPU using direct addressing. To solve this problem, the key-code is encoded as a 1D-texture in our system that can easily be transmitted to the shader program.

5 Methods

Mentioned techniques depicted in this section are for optimizing computations, for data visualization, and for volume rendering to help experimenters find CA functions in an efficient manner.

5.1 From 3D data set to single 2D texture

Before running the CA program, the 3D dataset is precomputed into a unique 2D buffer as shown in Figure 6. This buffer is stored in a portable network graphics (PNG) picture format consisting of a collection of slices from a density volume. This technique is convenient and efficient; it allows users to see the data set in a consistent way without requiring any additional software and the "volume image" can then be directly used as a texture for the *Fragment Shaders* program. Furthermore, due to the natural structure of pictures, accessing neighbors between cells is easy (but not trivial) and cellular interactions are therefore very efficient. Notice that to save useless import-export computations, this "flattening of the 3D volume" occurs at the input only, then along all GPU and CPU processes, only 3D volume (represented as 2D textures) is used. Also when storing the volume, a 3D PNG image is used as it can also be an input to the test bed.

To be able to compute a 2D texture from a 3D dataset using the graphics card in an efficient manner, we define a relationship between the edge size S_i of the volume image and the edge size S_v of the volume such that:

$$S_v^3 = S_i^2, \text{ with } \sqrt{S_v} \in \mathbb{Z}^+ \tag{3}$$

The first five couples of type "$[S_v, S_i]$" resulting from this equation are: $[2^2, 2^3]$, $[2^4, 2^6]$, $[2^6, 2^9]$, $[2^8, 2^9]$, $[2^{10}, 2^{15}]$. Notice that the last couple represents a voxel space of 10^9 cells, which is currently the most common MRI matrix size. However, due to memory issues, those very large matrices can be applied only to the latest graphics cards and their cellular interaction cannot yet run in real-time. For this reason, they are not further discussed in this paper.

5.2 Floating-point variation of CA

Also called coupled map lattice (CML), a floating point variation of CA [11] is a technique for the simulation of natural phenomena. Instead of using a discreet change of states, like Boolean, CML allows floating numbers as states. The truth table remains valid, since a unique threshold is defined. Nevertheless, using a derivative of *delta* instead of 0 or 1, CML permits intermediate steps (and therefore more natural visual effects) to occur. For some common functions, *i.e.* boiling, convection, reaction-diffusion processes, CML-CA and classical CA both converge to identical states. However, CML-CA

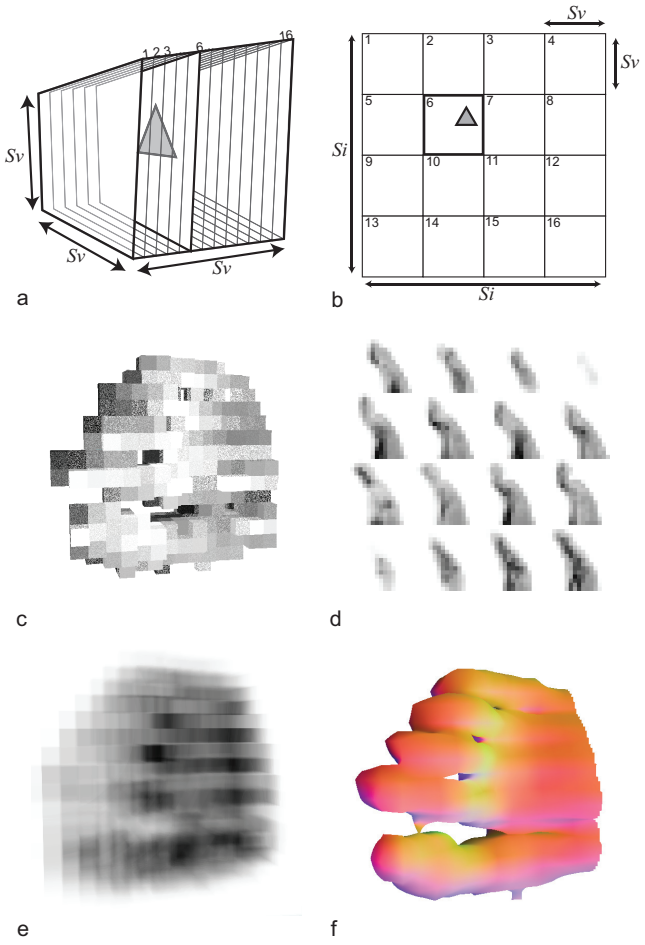

Fig. 6 Relationship between $3D$ matrices (a) and the equivalent $2D$ texture (b). As a practical example, rendered Figures c, d, e, and f illustrate a 16x16x16 matrix, and its 3D png, X-ray, and iso-surface representations respectively

distorts behaviors locally, then globally. While this distortion makes processes more complex, it also offers opportunities to explore new behaviors. The test bed presented in this study provides this powerful technique as an option. CML can be selected by using *non-Boolean step* on the user interface (Figure 1).

To help experimenters, a visual property in the *X-Ray like* simulation is included; this is further discussed in section 5.3.2. This property is directly derived from CML: coloring the derivative of the cellular density through time. When using floating point variation instead of direct Boolean change of state, the presented method provides a visualization of the cellular activity using color coding. That is, cells will turn red when density increases, and turn cyan when density decreases. This derivative can strongly help deduce crucial information required for defining CA emerging global behavior, as discussed in Section 6.

5.3 Rendering of 3DCA

The surface is often not the only important attribute as in medical visualizations [23]. For instance, finding effective volume rendering methods for interior (inside) of object (*e.g.* organs) remains a challenge in the field of computer graphics. This paper does not claim to overcome this challenge, however, to study emerging 3DCA behavior, the entire 3D cellular structure has to be visualized: the surface as well as the interior. For this reason, two types of rendering are proposed: X-Ray like simulation and sphere-map lighting surface rendering, including diffuse and specular sphere maps. The rendering must be in real-time; they are thus both implemented in a GPU Shader. Furthermore, to be able to understand and identified convenient CA, a simultaneous and partial X-ray and iso-surface rendering has been developed (see online provided software).

Figure 7 displays results from the two main types of *Fragment Shaders* algorithms that were developed to help identify global behaviors. The surface and the interior of the 3D cellular structure can be viewed on the two left images and two right images respectively. To illustrate the rendering differences, a close up of the same region is shown on the top left of each image. Following are the details of the advantages for both of these rendering techniques: X-ray like simulation for the interior, and sphere-map lighting surface rendering for the surface.

5.3.1 X-Ray like rendering

Seeing inside an object is not natural, however, at times tremendously useful. A common and well-known technique, mainly used in medicine, consists of casting X-Rays through biological tissues on an X-Ray sensitive film. To achieve an X-ray like visualization in our implementation, the algorithm illustrated in Figure 8 behaves in a similar way: the idea is to cast a ray from the isometric projection (A) of the camera position on the cube, through the 3D matrix (ending in B) and find an intensity I to sum s (for 'step') local densities i encountered such that:

$$I = \int_A^B i_x \delta_x \simeq \sum_{x=A}^{B} (\Delta_x . i_x), x \leftarrow x + \Delta_x \qquad (4)$$

with $\Delta_x \leftarrow 1/s$

For an optimum simulation, Δ_x should be the local distance from one voxel to another as the rays go through the volume. Figure 8(d) proposes a visually consistent simulation with only 50 steps.

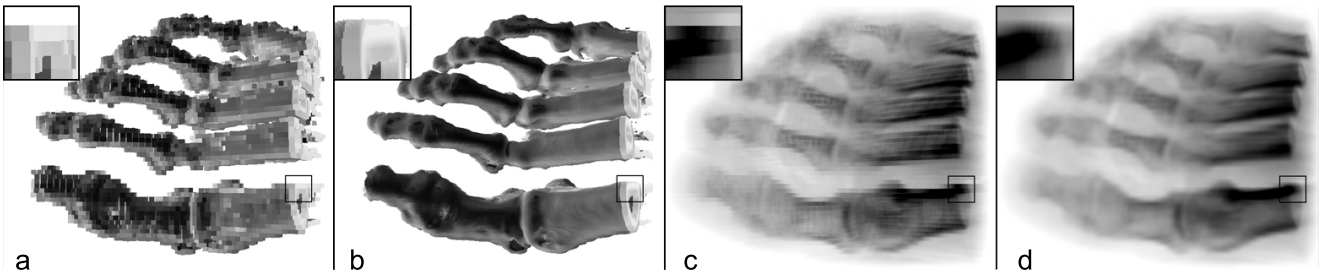

Fig. 7 Presenting the two rendering types, *i.e.* density surface and x-ray simulations, of the same 64^3 volume: (a) density voxel; (b) corresponding implicit surface; (c) X-ray like simulation on voxel; (d) supersampling X-ray like simulation

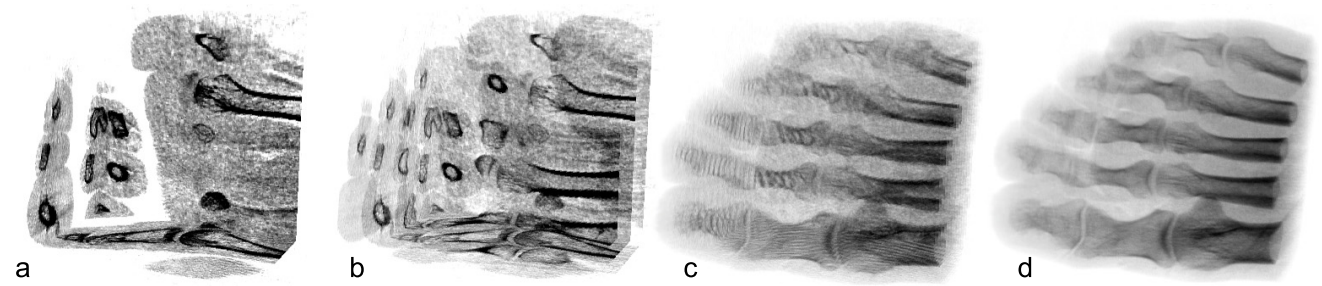

Fig. 8 Rendering quality on a 256^3 matrix, depending of the number of steps s for the X-ray simulation: (a) one step; (b) three steps; (c) 10 steps; (d) 50 steps

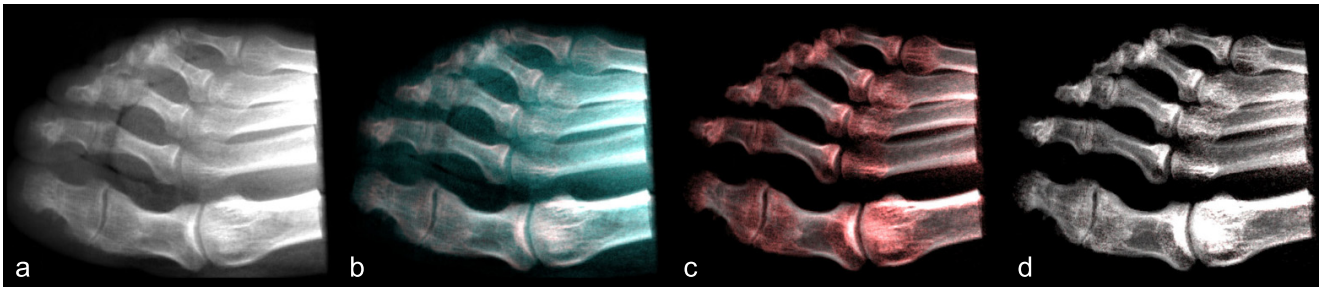

Fig. 9 CA tool in action using X-Ray visualization on a 256^3 voxel space: the derivative of local densities are shown in cyan when negative and red when positive

5.3.2 Adding colors to visualize cellular behavior

In Figure 9, a CA is applied that simulates the destruction of weak cells and the crystallization of dense ones: from left to right, the converging process from the source image to a stable state can be seen. The two intermediate images demonstrate the visual property derived from CML which is the color-coding described in section 5.2. The derivative of the cellular density is shown in red when positive and in cyan when negative. It can be observed that in the first phase, most cells lose density and in the second phase, the density of most remaining cells increases.

5.3.3 Surface rendering

To explain the selection of the rendering method, two remarks are necessary. First, it is difficult to define what is the surface of a 3D dataset consisting of densities that are often covered with noise. Second, the use of Boolean CA requires the use of a floating threshold (*e.g.* 0.5 for classical CA). Based on these two observations, a threshold, defining the "solidity" limit, was used for rendering, similarly to implicit surfaces [1]. The algorithm begins by casting a ray from the eye to find the first cell that has a density superior to the threshold. Then, to find the normal of the surface, the inverse sum of all normal vectors of existing surrounding cells multiplied by their respective densities was computed. Finally, as depicted in Figure 10, classic illumination techniques were applied. Figure 10 shows the results from the computation of diffuse and specular map texture coordinates and their respective sum with a sphere-map (double) lighting.

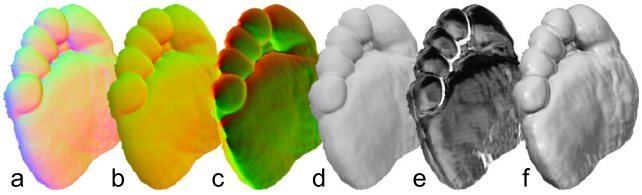

Fig. 10 Surface rendering pipeline: (a) normal computations, (b) diffuse sphere map texture coordinates, (c) specular sphere map texture coordinates, (d) and (e) corresponding diffuse and specular effects, (f) final combined rendering with sphere-map lighting

6 Results

This section first provides example visual outputs from different data types to demonstrate the flexibility of the method over different domains. Then it presents computational statistics for the presented CA method. Statistics were obtained through tests on four different hardware configurations. Finally, a discussion on the results is offered.

6.1 Graphical results and observations

CA are known to be efficient in a variety of application domains [27] ranging from mathematical physics [22] to urban studies and geography [17] and biological structures [18] to name a few. In this section, to demonstrate how the presented method behaves across some of these domains, sample results are presented for the following application fields: geometry, recursive patterns (fractals) and behavior (for bounding box and gradient), medical applications (visualization and cellular interactions), simulation of natural phenomena (vegetation growth-like and surface effects);

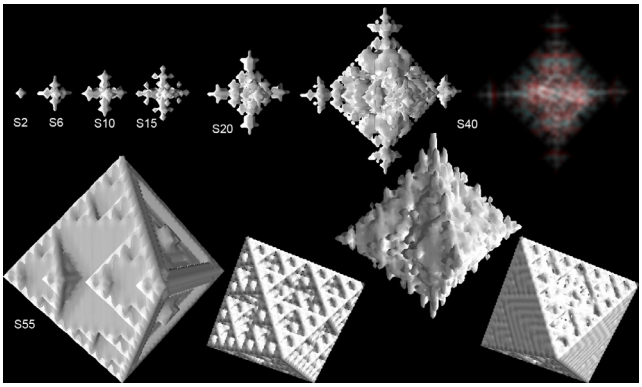

Fig. 11 An example of fractal produced from a single root cell

Figure 11 presents different fractal patterns generated from a single root using a low complexity symmetric CA key-code. On the first row, the growth process of a cross-like structure from steps 2 to 40 is presented. The last figure on the right illustrates the density view of the last state. Three triangular based fractal structures are shown on the bottom row at step 55 and the last two both depict step 40. The last image illustrates the effect of modifying a single symmetric CA parameter from the code of the recursive cross-like structure.

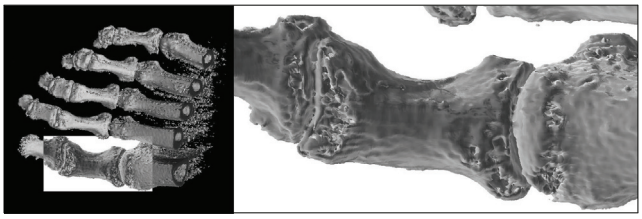

Fig. 12 An example for medical applications: real-time CA interactive effects and high quality data visualization on the topography of low density tissues (matrix of 256^3 voxels)

Figure 12 illustrates the potential of the method for medical applications. Once the CA function is determined, real-time interactive effects can be computed. In this example, destructive effects over low density tissues are computed and visualized. The close up view shows the quality of the final rendering for a 256 side cube matrix.

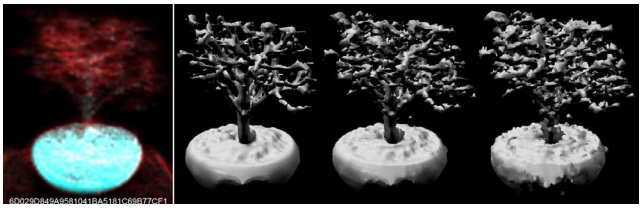

Fig. 13 An example for visualizing complex natural phenomena: vegetation simulation

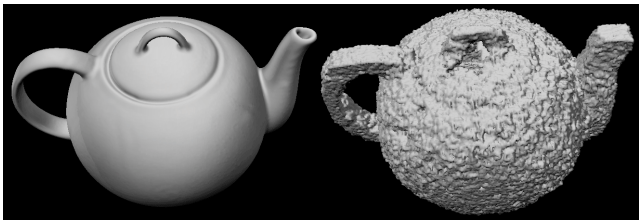

Fig. 14 An example for demonstrating complex surface effects (from left to right): original data (256^3 voxels) and wax-like simultaneous melting and growing

Figures 13 and 14 present examples of natural phenomena simulations: growth and wax surface. The first figure shows vegetation growth. In the first example, notice the X-Ray like simulation in the image on the

far left, two regions are clearly distinct: while density around the leaves visibly increases (reddish color), the density inside the pot decreases (cyan color). This particular CA seems to simulate the vegetation growth process. In reality, this CA only decreases saturated density regions and increases space regions with high gradient. Concerning the wax surface simulation, this surprising CA modified the surface of the initial teapot so that the object seems to be covered by a wax layer.

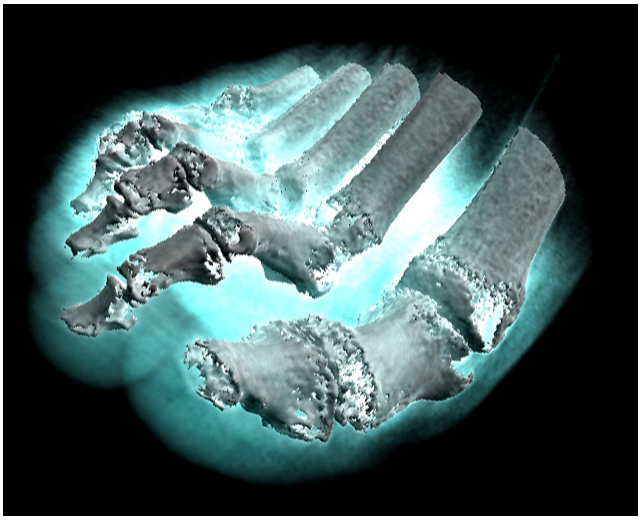

Fig. 15 Double visualization effects –*i.e.* X-Ray like and surface reconstruction– computed in a single rendering; the cyan effect represents the decreasing intensity flesh tissues: corresponding cells are programmed with a very simple cellular automaton that consists of spreading local densities towards zero or one with a predefined threshold

The real-time computation and visualization of CA is powerful, especially when understanding of complex datasets is necessary. In medical applications, for instance, real-time visualization of large datasets such as MRI scans is a challenge, and remains impossible on low-cost systems. Our method can be of help in some of these cases as it can handle a grid of 256^3. To our knowledge, the best MRI grids are up to 1024^3, thus, our method is close to allowing very low cost 3D visualization for such applications. More than a tool to only visualize in real-time, our approach also allows for the processing of complex operations in real-time using pre-determined CA functions (such as intelligent organ detection, or growth probabilistic expectations, for instance). Different kinds of potentially useful biomedical images can be seen in various figures of this paper. In particular, Figure 15 shows an X-ray like simulation simultaneously with implicit surface reconstruction.

Table 3 Four hardware configurations were tested; **Cfg, CT-OS** stands for configuration number, computer type and operating system, and **PC**, **NB**, **XP**, **Vst**, **7**, **Qd**, and **GF**, stand for personal computer, notebook, Windows-XP professional, Windows-Vista, Windows-7, NVidia, Quadro, and GeForce, respectively.

Cfg, CT-OS	Graphics Card	Processor and RAM
#1, NB-XP	Qd-FX1500M, 512MB	T7200, 1.99GHz, 1008MB
#2, PC-Vst	GF-8600-GS, 512MB	Q6600, 2.39GHz, 3072MB
#3, NB-XP	GF-7900M-GTX, 512MB	T7200, 2.00GHz, 2048MB
#4, PC-7	GF-295-GTX, 896MBx2	i7-920, 2.67GHz, 6144MB

Table 4 Performance comparison for four different PCs configurations: number of CA operations per second (in millions) –see also resulting ratio Figure 16

	CPU			GPU		
Matrices:	16^3	64^3	256^3	16^3	64^3	256^3
Config.#1	42.35	29.48	4.62	40.65	105.03	11.26
Config.#2	57.87	30.63	4.25	37.30	69.63	37.47
Config.#3	46.50	24.83	3.59	10.63	163.77	283.01
Config.#4	30.83	17.17	7.44	60.99	793.11	1731.04

6.2 Performance

In this section, results are presented for acceleration of CA computations by comparing different types of CPU or GPU based algorithms, different hardware configurations (four different graphics cards), and different data volumes (*i.e.* 16^3, 64^3, and 256^3).

6.2.1 Four configurations

All algorithms presented in this paper were developed in $C++$ on Microsoft Windows XP-pro using $MSVC++$, the graphics library OpenGL [20], and the OpenGL Shading Language ($GLSL$) [19] –some pseudo-codes are presented in the Appendix, (Figures 17 and **??**. To document the efficiency of the method, its performance on common everyday computers is reported. Advantages and limitations of using graphics cards is demonstrated, in particular for 3D. Generation of the graphics card is a parameter. Table 3 presents the four hardware configurations used for testing the presented method including two notebooks, and two PCs, four different processors and four different graphics cards.

6.2.2 GPU and CPU results

Figure 16 illustrates the average performance for the presented method. Reported statistics are obtained by showing the computation time for 10 thousand iterations. This was done on the three possible different buffer sizes, shown in the red, green, and blue zones.

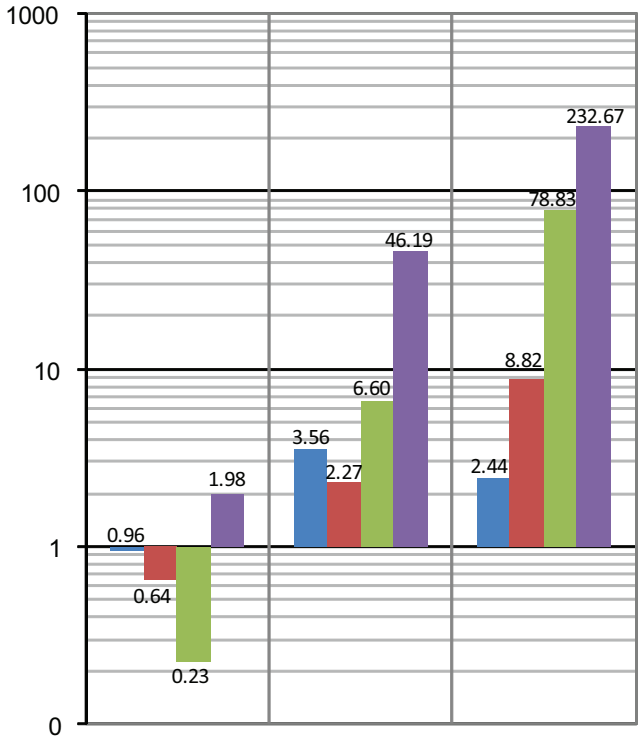

Fig. 16 *GPU/CPU* ratio of average performance based on values in Table 4 respectively for the four hardware configurations and the three matrix types (16^3, 64^3, and 256^3).

Table 4 presents the number of state changes (in millions) that are computed (using CPU and GPU) every second on the three possible 3D matrices, representing 4096 cells, about 262 thousands cells, and almost 17 million cells. The first observation is that the performance of all the CPUs quickly decrease in a very similar way as the matrices grow. The second observation is that GPU performances remain more or less identical for the 7000-series and equivalent and increase strongly for the newer series.

The chart depicted in Figure 16 displays the ratio between respective GPU and CPU capabilities. To improve its readability, a logarithmic scaling was used. Three observations can be made with regard to this chart. First, the ratio is inferior to one, there is no point in using the GPU for small 3D matrices. Second, for matrices larger than 16^3, graphics cards equivalent to or newer than the GeForce 8800GTX not only give excellent results, but demonstrate a real gap in terms of computation compared to older generations. Third, the excellent $\times$232 ratio is obtained with the latest graphics card, and even using best multi-core CPU with multi-thread algorithm, a minimum of $\times$30 is expected when GPU is used.

These results open real possibilities for low cost real-time massive visualizations and cellular interaction of advanced computations. For instance, the medical domain needs very high resolution imagery implying the use of very large matrices. As previously said, this is also a domain where intelligent 3D computation tools (such as organ detection, surgery training, tumor statistical growth prediction) are more and more needed.

7 Conclusions and future work

This paper presented a novel approach to simulating *bottom-up* 3D CA using the GPGPU bringing together a set of methods. A summary of key contributions can be listed as: transferring any CA computational key-code onto the graphics card, creating a model to do the computations for three dimensional CA, and an interactive interface (using simultaneously X-ray-like and iso-surface renderings) that helps finding convenient CA behavior. The interface is also designed to serve as a test bed for performance testing. As a result, faster CA simulations and much more efficient identification and classification are possible. Furthermore, a critical discussion was provided on two distinctly different approaches to CA based on the literature, namely *top-down* and *bottom-up* approaches.

After introducing the programming techniques using both CPU and GPU, and summarizing concepts used in this application, a novel method was proposed to encode large CA key-codes allowing a generalized Boolean CA algorithm to be performed on a GPU, restricted to memory size limitations. Following that, an original method to automatically sort symmetric CA (for any dimension) based on their symmetric structure was presented. A detailed account was provided as to how to encode generalized algorithms for $3D_{vN}CA$. To demonstrate the capabilities and flexibility of the model, examples of characteristic patterns and common global behaviors were presented. Computational statistics obtained on different hardware and software configurations demonstrated a very strong performance gain on common personal computers, especially for large 3D datasets. It was also demonstrated that the current voxel model behaves in a different way in terms of computational capabilities depending on the graphics card generation. The solutions that are offered in this paper will allow for a true *bottom-up* CA key-code (rule) discovery for potential users of CA and researchers across many fields.

Following the work described in this paper, in the near future we plan to report the findings on an extended model for other three-dimensional Boolean CA with indirect neighbors, such as Moore models $3D_b^{M_1}CA$ and $3D_b^{M_2}CA$ (18 and 26 neighbors). Also, we believe,

harmonious structures such as both hexagonal and cubic close packing $3D^hCA$ models [16] are the most promising field of study for the 3D CA, with only 12 neighbors and the most natural structure. Nevertheless, changing the geometric nature of CA (from cubic to 3D hexagonal) will imply strong changes in the algorithm. Therefore, direct integration in the system proposed in this paper is not possible. Finally, exploring possibilities of new visualizations for this model in stereoscopic virtual environments, such as a CAVE with haptic interaction, is currently being considered.

Appendix

In this appendix, pseudo codes of the GLSL pixel-shaders for the CA rule decoding algorithm (Figure 17) and the iso-surface rendering (Figure 18) are presented.

Acknowledgments

We are grateful to Dr. Helena Grillon, Dr. Junghyun Ahn, and Mr. Patrick Salamin for their suggestions, which greatly improved the manuscript. We also would like to express our gratitude to the three anonymous reviewers for their valuable feedback. This work was supported by the following grants: SNF grant "GeoF" (no. 120434), SNF grant "AERIAL**CROWDS**" (no. 122696), and the European Union COSI-ICT "CYBER-**EMOTIONS**" (IST FP7 231323).

References

1. Bloomenthal, J., Bajaj, C., Cani, M.P., Rockwood, A., Wyvill, R., Wywill, G.: Introduction to Implicit Surfaces. Morgan Kaufmann, San Francisco, CA, USA (1997)
2. Çöltekin, A., Haggrén, H.: Stereo foveation. The Photogrammetric Journal of Finland **20**(1) (2006)
3. Conway, J.: Game of life. Scientific American **223**, 120–123 (1970)
4. Coutinho, B.B.S., Giraldi, G., Apolinario, A., Rodrigues, P.: GPU surface flow simulation and multiresolution animation in digital terrain models. In: LNCC Reports 2008. Petrpolis/RJ, Brasil (2008)
5. Duchowski, A., Çöltekin, A.: Foveated gaze-contingent displays for peripheral lod management, 3d visualization, and stereo imaging. ACM Transactions on Multimedia Computing, Communications, and Applications (TOMCCAP) archive **3**(6) (2007)
6. Ganguly, N., Sikdar, B.K., Deutsch, A., Canright, G., Chaudhuri, P.P.: A survey on cellular automata. Tech. rep., Centre for High Performance Computing, Dresden University of Technology (2003)
7. Gardner, M.: Mathematical games: The fantastic combinations of john conway's new solitaire game "life". Scientific American (1970)
8. Gardner, M.: Mathematical games: On cellular automata, self-reproduction, the garden of eden and the game "life". Scientific American (1971)

```
-------------------------------------------------------------------------------
0. Variables declaration
0.1 Declaration of the information transmited from the C++ code
. the 3D CA voxel space stored as a 2D texture: 'source_tex'
. the CA code as a 1D texture: 'code_3DvNCA_tex'
. the voxel space cube side size, that can be 16, 64, or 256: 'cubeSide'
Options:
. a CA threshold defining the Boolean limit: 'threshold' default been 0.5
. a non Boolean change of state: 'nonBoolCAStep' default been 0.5
-----
0.2 Declaration of global variables
. texture side size (64, 512, or 4096): 'texSide' <- cubeSide^1.5
. group number from 2D to 3D (4, 8, or 16): nbOfGroup <- texSide / cubeSide
-----
0.3 Declaration of local variables
. cell 2D position: 'P_coord' <- gl_TexCoord[0].st
. 3D position: 'P_3DVoxel' <- 3DCoordFromfloat2DCoord( P_coord )
. current (P) and neighbor cells state: 'A,B,C,D,E,F,P' all equal to 0
. real value for the CA code: 'ca_code'
. vector 2D to get the AC code coordinate: 'code_coord'
. final texel 4d color vector to define: all param 'RGBA' <- 1.0
-------------------------------------------------------------------------------
1. General code
  1.1 Getting current (P) cells state
  if texture2D(source_tex, P_coord.st).r sup. to threshold then P <- 1
  endif

  1.2 Getting neighbor cells state
  - check CA grid limit in (x) for A and B:
    if P_3DVoxel.x inferior to cubeSide-1 then
      => define A coordinates 'A_coord' and assign the result of:
         2DCoordFrom3DVoxelCoord( P_3DVoxel + ivec3(1,0,0) )
      => if texture2D( source_tex, A_coord.st ).r sup. threshold then A <- 1
         endif
    endif
    if P_3DVoxel.x superior to 0 then
      => define B coordinates 'B_coord' and assign the result of:
         2DCoordFrom3DVoxelCoord( P_3DVoxel + ivec3(-1,0,0) )
      => if texture2D( source_tex, B_coord.st ).r sup. threshold then B <- 1
         endif
    endif
  - check CA grid limit in (y), for C and D:
    if( P_3DVoxel.y < cubeSide-1 ) then
      => define C coordinates 'C_coord' and assign the result of:
         2DCoordFrom3DVoxelCoord( P_3DVoxel + ivec3(0,1,0) )
      => if texture2D( source_tex, C_coord.st ).r sup. threshold then C <- 1
         endif
    endif
    if( P_3DVoxel.y > 0 ) then
      => define D coordinates 'D_coord' and assign the result of:
         2DCoordFrom3DVoxelCoord( P_3DVoxel + ivec3(0,-1,0) )
      => if texture2D( source_tex, D_coord.st ).r sup. threshold then D <- 1
         endif
    endif
  - check CA grid limit in (z), for E and F:
    if( P_3DVoxel.z < cubeSide-1 ) then
      => define E coordinates 'E_coord'and assign the result of:
         2DCoordFrom3DVoxelCoord( P_3DVoxel + ivec3(0,0,1) )
      => if texture2D( source_tex, E_coord.st ).r sup. threshold then E <- 1
         endif
    endif
    if( P_3DVoxel.z > 0 ) then
      => define F coordinates 'F_coord' and assign the result of:
         2DCoordFrom3DVoxelCoord( P_3DVoxel + ivec3(0,0,-1) )
      => if texture2D( source_tex, F_coord.st ).r sup. threshold then F <- 1
         endif
    endif
-------------------------------------------------------------------------------
  2. Defining the CA code and its coordinate:
  ca_code    <- (P + 2A + 4B + 8C + 16D + 32E + 64F) / 128
  code_coord <- vectorCoordinateIn2D( ca_code, 0 )
-------------------------------------------------------------------------------
  3. CA assignment:
  if pass sup. to threshold then next_state <- true
  endif
  RGBA.xyz <- texture2D( source_tex, P_coord.st ).rgb
  if next_state is true
    then
      => RGBA.r <- texture2D( source_tex, P_coord.st ).r + nonBoolCAStep
      => if RGBA.r sup. to 1 then all rgb values of RGBA <- 1
    else
      => RGBA.r <- texture2D( source_tex, P_coord.st ).r - nonBoolCAStep
      => if RGBA.r inf. to 0 then all rgb values of RGBA <- 0
  endif
-------------------------------------------------------------------------------
  4. Final texelcolor assignment:
  gl_FragColor <- RGBA
End General code
-------------------------------------------------------------------------------
```

Fig. 17 Fragment Shader used as a GPGPU: cellular automaton pseudo code algorithm

9. Gerhardt, M., Schuster, H., Tyson, J.: A cellular automaton model of excitable media including curvature and dispersion. Science-247 **46**, 1563–1566 (1990)
10. Gobron, S., Bonafos, H., Mestre, D.: GPU accelerated computation and visualization of hexagonal cellular automata. In: H. Springer Berlin (ed.) 8th International Conference on Cellular Automata for Research and Industry, ACRI 2008, vol. 5191/2008, pp. 512–521. Yokohama, Japan (2008)
11. Harris, M.J., Coombe, G., Scheuermann, T., Lastra, A.: Physically-based visual simulation on graphics hardware. In: HWWS'02: Proceedings of the ACM SIG-

```
-------------------------------------------------------------------------------
0. Variables declaration
0.1 Declaration of the information transmited from the C++ code
. the voxel space cube side size, that can be 16, 64, or 256: 'cubeSide'
0.2 Declaration of the information transmited from vertex shader
. a 3D vector 'CameraPosition'
-------------------------------------------------------------------------------
1. specific rendering pseudo-de
Define 'renderTheSurface' function with parameters
    real 'distance', 'step'
    vector 3D 'finalPos', 'eyeDirection'
{
  define two vectors 4D 'RGBA' and 'localColor' both assigned to ( 0, 0, 0, 0 )
  define two vectors 3D 'xyz' and 'texelCoords3D' both assigned to ( 0, 0, 0 )

  define a vector 2D called 'texCoordIn2D'

  1.1 find the first voxel to be seen and get its color in RGBA
  define a Boolean 'OK'  <- false
  define a real 'rayPos' <- 0
  while rayPos < distance and not OK
    xyz           <- finalPos + rayPos * eyeDirection
    texelCoords3D <- xyz * float( cubeSide )
    texCoordIn2D  <- float2DCoordFrom3DVoxelCoord( texelCoords3D )
    localColor    <- texture3D( sourceTex4, texelCoords3D )
    if localColor.a sup. to threshold then
      => OK <- true
      => stop loop
    endif
    rayPos += step
  endWhile
  if not OK then nothing to be rendered => stop here

  1.2 determine the normal vector "n" depending of the 28 neighbors Ni
  vec3 n = determineNormal( texelCoords3D );

  1.3 finding the best camera image plan for accurate color recovery
  define table of five real called 'dotCi' such that for any k E {1 to 5}:
    dotCi[k] = clamp( dot( camLookAt{k}, n ), -1.0, 0.0 )
  define table of five vector 4D called 'CiInfluence' such that:
    CiInfluence[k] = texture3D( sourceTex(k), texelCoords3D )
  define table of five real called 'minDotCi' such that:
    minDotCi[k] <- 0.0
  define real 'minDot' <- dotCi[0]
  define integer 'minDotIndex' <- 0
  startLoop( integer 'i' <- 0; condition (i < 5); increment i )
    if dotCi[i] < minDot then
      => minDot <- dotCi[i]
      => minDotIndex <- i
    endIf
  endLoop
  minDotCi[minDotIndex] <- 1.0
  startLoop( integer 'i' <- 0; condition (i < 5); increment i )
    RGBA.rgb is decreaded of the result of {dotCi[i] . CiInfluence[i] . minDotCi[i]}
  endLoop
  RGBA.a <- 1
  return( RGBA )
End 'renderTheSurface'
-------------------------------------------------------------------------------
2. Main code
  define vec.4D 'finalColor' <- vec4( 0.0 )
  define vec.3D 'eyeDirection' <- normalize( vec3( gl_ModelViewMatrixInverse
                                  . vec4( normalize( CameraPosition ), 0.0 ) ) )
  define real parameter 'distance' <- very large value
-----------------------------------
  2.1 testing clipping plans to determin appropriate distance value:
  2.1.1 collision with positive and negative X-plan
  if X value of the eyeDirection superior to zero
    then distance <- min( distance, ( 1.0 - vertex.x ) / eyeDirection.x )
    else distance <- min( distance, ( 0.0 - vertex.x ) / eyeDirection.x )
  endif
  2.1.2 collision with positive and negative Y-plan
  if Y value of the eyeDirection superior to zero
    then distance <- min( distance, ( 1.0 - vertex.y ) / eyeDirection.y )
    else distance <- min( distance, ( 0.0 - vertex.y ) / eyeDirection.y )
  endif
  2.1.3 collision with positive and negative Z-plan
  if Z value of the eyeDirection superior to zero
    then distance <- min( distance, ( 1.0 - vertex.z ) / eyeDirection.z )
    else distance <- min( distance, ( 0.0 - vertex.z ) / eyeDirection.z )
  endif
-----------------------------------
  2.2 Ray tracing from the end point to the start point,
     in order to properly manage transparency:
  define real parameter 'step' <- 1.0 / ( cubeSide . Precision )
-----------------------------------
  2.3 Final texelcolor assignment:
  gl_FragColor <-  renderTheSurface( distance, step, vertex, eyeDirection )
End Main code
-------------------------------------------------------------------------------
```

Fig. 18 Fragment Shader used for rendering: finding surface normal and reconstructing corresponding surface texture pseudo code algorithm

GRAPH/EUROGRAPHICS conference on Graphics hardware, pp. 109–118. Eurographics Association, Aire-la-Ville, Switzerland, Switzerland (2002)

12. L. Durbeck, N.M.: The cell matrix: An architecture for nanocomputing. Nanotechnology **12**, 217–230 (2001)
13. M. Tomassini M. Sipper, M.P.: On the generation of high-quality random numbers by two-dimensional cellular automata. IEEE Transactions on Computers **49**(10), 1146–1151 (2000)
14. von Neumann, J.: Theory of Self-Reproducing Automata, chapter in Essays on Cellular Automata. University of Illinois Press, Urbana, Illinois (1970)
15. Owens, J., Luebke, D., Govindaraju, N., Harris, M., Krueger, J., Lefohn, A.E., Purcell, T.J.: A survey of general-purpose computation on graphics hardware. Computer Graphics Forum **26-1**, 80–113 (2007)
16. Phillips, W., Phillips, N.: An Introduction to Mineralogy for Geologists. John Wiley & Sons (1981)
17. Pinto, N., Antunes, A.: Cellular automata and urban studies: A literature survey. ACE: architecture, city and environment **1**(3) (2007)
18. Prusinkiewicz, P.: Modeling and visualization of biological structures. In: Proceeding of Graphics Interface 93, pp. 128–137. Toronto, Ontario, Canada (1993)
19. Rost, R.: OpenGL Shading Language. Addison Wesley Professional, 2nd edition (2006)
20. Shreiner, D., Woo, M., Neider, J., Davis, T.: OpenGL Programming Guide: the official Guide to learning OpenGL v2.0. AddisonWesley Professional, 1st edition (2005)
21. Singler, J.: Implementation of cellular automata on a GPU. In: ACM Workshop on General Purpose Computing on Graphics Processors, sponsored by ACM SIGGRAPH. Los Angeles, USA (2004). URL http://www.jsingler.de/studies/cagpu.php
22. Smith, M.: Cellular automata methods in mathematical physics. Ph.D. thesis, Institut National Polytechnique de Grenoble (1994)
23. Tatarchuk, N., Shopf, J., DeCoro, C.: Advanced interactive medical visualization on the GPU. Journal of Parallel and Distributed Computing **68**(10), 1319–1328 (2008)
24. Thalmann, D.: A *lifegame* approach to surface modeling and rendering. The Visual Computer **2**, 384–390 (1986)
25. Tran, J., Jordan, D., Luebke, D.: New challenges for cellular automata simulation on the GPU. www.cs.virginia.edu (2003)
26. Wolfram, S.: Universality and complexity in cellular automata. Physica D **10**, 1–35 (1984)
27. Wolfram, S.: A new kind of science. Wolfram Media Inc., 1st edition (2002)
28. chu Wu, T., yi Hong, B.: Simulation of urban land development and land use change employing gis with cellular automata. In: Second International Conference on Computer Modeling and Simulation, pp. 513–516. Institute of Electrical and Electronics Engineers (2010)
29. Zaloudek, L., Sekanina, L., Simek, V.: Gpu accelerators for evolvable cellular automata. In: Computation World: Future Computing, Service Computation, Adaptive, Content, Cognitive, Patterns, pp. 533–537. Institute of Electrical and Electronics Engineers (2009)
30. Zhao, Y.: GPU-accelerated surface denoising and morphing with lattice boltzmann scheme. In: IEEE International Conference on Shape Modeling and Applications (SMI'08). Stony Brook, New York, USA (2008)